The Creation of a CPU Timer for High Fidelity Programs

Aidan A. Dick[1]
NASA Marshall Space Flight Center, Huntsville, AL, 35812

Using C and C++ programming languages, a tool was developed that measures the efficiency of a program by recording the amount of CPU time that various functions consume. By inserting the tool between lines of code in the program, one can receive a detailed report of the absolute and relative time consumption associated with each section. After adapting the generic tool for a high-fidelity launch vehicle simulation program called MAVERIC, the components of a frequently used function called "derivatives ( )" were measured. Out of the 34 sub-functions in "derivatives ( )", it was found that the top 8 sub-functions made up 83.1% of the total time spent. In order to decrease the overall run time of MAVERIC, a launch vehicle simulation program, a change was implemented in the sub-function "Event_Controller ( )". Reformatting "Event_Controller ( )" led to a 36.9% decrease in the total CPU time spent by that sub-function, and a 3.2% decrease in the total CPU time spent by the overarching function "derivatives ( )".

Nomenclature

CPU	=	Central Processing Unit
MAVERIC	=	Marshall Aerospace Vehicle Representation in C
μ	=	Average
σ	=	Standard Deviation
Windows	=	Microsoft Windows Series
Red Hat	=	Red Hat Enterprise Linux Release 6.1

I. Introduction

Some members of the Flight Mechanics and Analysis Division (EV40) at NASA use a program called MAVERIC to collect flight data. MAVERIC is a high fidelity launch vehicle simulator that uses computational integration to calculate the trajectory of a vehicle. The state of the vehicle is updated two hundred times per simulated second, which requires that various functions be called at that same rate, resulting in tens of thousands of function calls per MAVERIC run.

During the design phase of launch vehicle development, EV40 makes extensive use of MAVERIC. During this phase, MAVERIC itself is run hundreds of thousands of times. Small increases in the speed of MAVERIC can result in hours of decreased simulation run time. Inversely, functions that use excess CPU time result in hours of wasted time spent waiting for simulations to run. Keeping each aspect of MAVERIC efficient is essential to maintaining productive use of time in the EV40 division.

To help in efforts towards maintaining MAVERIC's efficiency, a generic tool to measure CPU time was created. The tool was then adapted specifically for MAVERIC's output system. Employing such a tool reveals which functions are the most costly, where improvements can be made, and how effective your improvements are.

II. Background

MAVERIC is stored and executed on a shared server, which is connected to by members of EV40 using the Windows program X-Win32. The server operates using the Linux-based operating system called Red Hat. Most Linux and UNIX systems have built in time functions that can be called by programs. The most common function in C, "clock ( )", is too imprecise to be used for high fidelity programs. In Red Hat, there is a built in function called

[1] NASA Marshall Space Grant Research (MSGR) Intern, Space Craft & Vehicle Systems Department, Marshall Space Flight Center, University of Minnesota – Twin Cities.

"clock_gettime ( )", which provides access to several useful timers, some of which have the resolution of nanoseconds (Rutenberg, 2007).

III. Technical Approach

The concept for the CPU timer was simple, but the design had to meet several important requirements. To use the CPU timer, the timer is placed in the program's code. When the program runs with the probed code, the CPU function makes measurements, stores the data, and then reports the data in a consolidated, easily read format.

A. Wall Time vs. CPU Time

One concern to address when creating a CPU timing function is the distinction between the 'wall-clock time' and the CPU time. The wall-clock time is a measurement of the total time that elapses between two points in the code. Wall clock-time includes process time, communication channel delay, and other programmed delays. The delays that effect the wall-clock time are independent of the code, and therefore should not be taken into consideration. In order to accurately measure the program's code, the only thing that should be measured is the CPU time, which can be performed by using the UNIX function "clock_gettime ( )". This UNIX function fills a previously declared struct, a storage type in C and C++ that combines a set of objects into a single object, with the current CPU timestamp. To record the amount of CPU time a specific function takes, the "clock_gettime ( )" function needs to be called twice. It should be called once right before the timed section of code, to record the timestamp at the beginning, and once right after the timed section of code, to record the timestamp at the end. The amount of CPU time the process took is equal to the difference in the two timestamps.

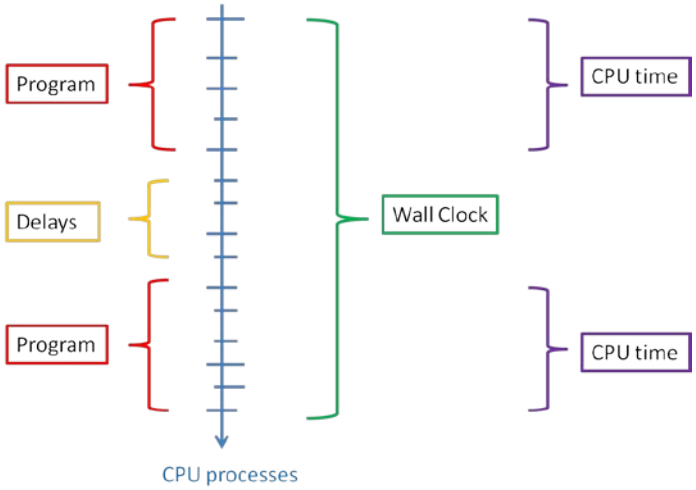

Figure 1. The Difference between Wall Clock and CPU Time. *A process timeline is labeled according to process type on the left, and is labeled according to timer type on the right.*

B. Familiarizing with "clock_gettime ( )"

Before implementing the clock function in a program, it's important to first familiarize with its capabilities. After testing the clock function in various smaller programs, it became evident the timer takes a notable amount of CPU time to run itself. The two CPU timestamps recorded when two "clock_gettime ( )" functions are placed one after the other have a difference of 1.3-1.8 million nanoseconds. This is a small, but not entirely negligible amount of CPU time. This is a limiting factor in the resolution error of the CPU timer used for MAVERIC.

C. Creating the Structure for the Timer

Many complex programs use functions that are called multiple times. In order to accurately measure the total CPU time used by a particular function, or a particular section of code, it's important to have the ability to store multiple timestamps. For ease of use, the function should also be capable of handling multiple timers at once. Addressing both of these requirements (i.e., handling multiple function calls and handling multiple timers), the function uses a vector system to store data.

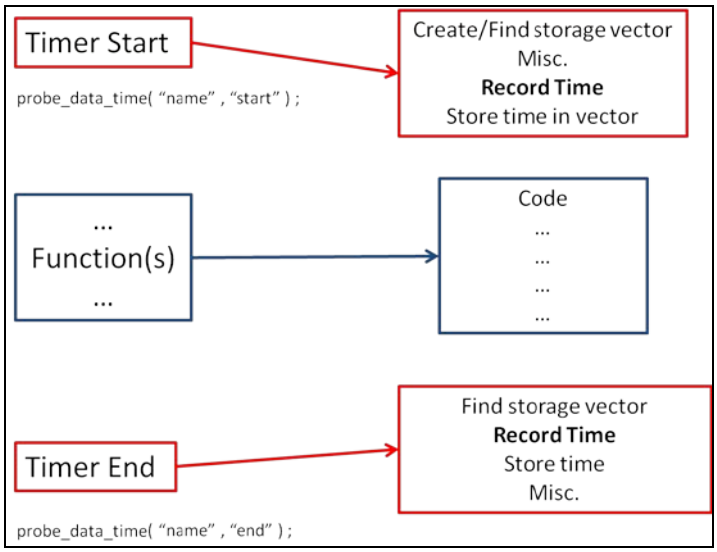

Figure 2. The Timer Process. *On the left, a start timer and an end timer (in red) are inserted before and after the section of code to be timed. On the right are the sub-processes that are executed when each line on the left is called.*

In order for the program to distinguish between different timers, each timer is given a name in the implemented code. The timer is also given a position statement (i.e., start or finish) as shown in Figure 2 and Figure 3. When the timer is called, it uses those two pieces of information. If it is the first time that specific timer has been seen by the program, it creates a new vector to store the data. If the program has seen that timer's name before, it records the current timestamp and stores it in its vector. After the data struct in the vector has both a start time and an end time, it subtracts the difference between the two times, and stores the value, which corresponds to the CPU time used by that section of code. This process is repeated each time that section of code is executed, and is not limited by any storage value.

In order to ensure that the time recorded best reflects the true CPU time for the timed section of code, the Unix function that records the timestamp is called as close to the program's code as possible, as seen in Figure 2. This ensures that the majority of the CPU time recorded corresponds to the timed section of program code, and doesn't correspond to the code for the timer itself.

To keep the [word for used memory] at an appropriate level, the CPU timer consolidates its data and stores it to file at various check points throughout program execution.

D. Using the Timer

To time a function or section of code within a program, the timing function "probe_data_time ( )", needs to be placed once before the timed section, and once after the timed section. The timer also requires two parameters to be given. The first function parameter to be given is the name that will be assigned to the section of code, and the second parameter is whether it is the "start" or "end" of the timed section of code.

```
probe_data_time( "name" , "start" );

probe_data_time( "name" , "end" );
```

Figure 3. The Start and End Probes. *These lines of code are inserted before and after the segments of code that are to be timed by the CPU timer.*

Maintaining that each timer has a matching pair for every "start" or "end" timer is critical for ensuring that each timer measures accurately. Timer pairs that are placed on different sides of "for" and "while" loops or "if" and "else" statements all have potential to produce incorrect CPU times.

E. The "Relative" Timer

A special timer was also developed to be compared by the other individual timers. This timer was named the "relative" timer. When the program is finished running, the total values for each timer are calculated. The individual total times are then calculated as a percentage of the "relative" timer, if the "relative" timer has been implemented, and are reported alongside their total values, as shown in Figure 5.

This feature allows users to set a timer as a reference point to compare data to. By reading CPU time as a percentage of another function, as opposed to in total CPU time format, users can more intuitively develop an understanding of the results.

F. Error in the Timer

As discussed, the function "clock_gettime ( )" requires a small but significant amount of CPU time to be processed. When multiple timers are placed within another timer, such as the "relative" timer, small gaps appear in the timed sections, as shown in Figure 4. These gaps represent CPU time being measured by various timers that do not correspond to CPU time used for processing MAVERIC's code, but CPU time used for processing the timers themselves.

Because this timer is to be used as a utility for better understanding the code, and not for precise measurements, a small resolution error is acceptable.

G. Reporting the Information

Once the program has finished executing, the results are consolidated into a report file. After gathering data that has been previously stored to file, the program ranks each timer by size. The program then reports each timer's data in order of size, as shown in figure 5.

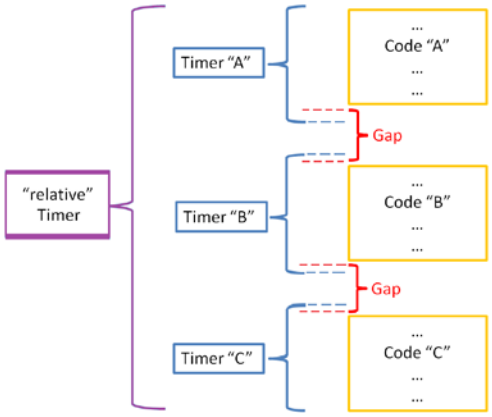

Figure 4. The Resulting CPU Time Gaps. *Due to CPU time spent processing the timer's commands; empty CPU gaps appear, shown between the dotted lines.*

File Edit Search Preferences Shell Macro Windows

	Line_Number	Probe_Name	Total_Time	Relative_Percent	Ind_Comp_Percent	Count
2	2	relative	6.937119e+10	100	N/A	49665
3	3	flex	1.665202e+10	24.0042	26.149	49665
4	4	mass_prop	7.515790e+09	10.8342	11.8022	49665
5	5	aero	5.850042e+09	8.43296	9.18643	49665
6	6	event_controller	5.698495e+09	8.2145	8.94845	49665
7	7	propulsion	5.323820e+09	7.6744	8.36009	49665
8	8	rcs	4.751961e+09	6.85005	7.46209	49665
9	9	sensors	3.478276e+09	5.01401	5.462	49665
10	10	traj_state	3.377881e+09	4.86928	5.30435	49665
11	11	fsw	2.207699e+09	3.18244	3.46679	49665
12	12	slosh	1.418396e+09	2.04465	2.22733	49665
13	13	environment	1.060945e+09	1.52937	1.66602	49665
14	14	actuators	1.057304e+09	1.52413	1.6603	49665
15	15	eom	8.334441e+08	1.20143	1.30877	49665
16	16	update_time	6.806022e+08	0.981102	1.06876	49665
17	17	tail_wag	6.524388e+08	0.940504	1.02454	49665
18	18	track_assets	5.539500e+08	0.79853	0.869877	49665
19	19	bomb_out	3.848958e+08	0.554835	0.604409	49665
20	20	config	2.682938e+08	0.386751	0.421307	49665
21	21	planets	2.121259e+08	0.305784	0.333105	49665
22	22	liftoff	1.593619e+08	0.229723	0.250249	49665
23	23	sep	1.508487e+08	0.217452	0.23688	49665
24	24	gen	1.335932e+08	0.192577	0.209784	49665
25	25	prop_veh_list	1.219553e+08	0.175801	0.191509	49665
26	26	pu	1.204553e+08	0.173639	0.189153	49665
27	27	structs	1.113732e+08	0.160547	0.174891	49665
28	28	loads	1.054655e+08	0.152031	0.165614	49665
29	29	actuator_fail	1.051656e+08	0.151598	0.165143	49665
30	30	grav_grad	1.050364e+08	0.151412	0.164941	49665
31	31	therm_prop	1.042379e+08	0.150261	0.163687	49665
32	32	stress	1.011857e+08	0.145861	0.158894	49665
33	33	landing_gear	1.001468e+08	0.144364	0.157262	49665
34	34	dock_ports	9.704182e+07	0.139888	0.152386	49665
35	35	misc	9.615202e+07	0.138605	0.150989	49665
36	36	exthlf	9.098542e+07	0.131157	0.142876	49665
37						

Figure 5. A Screenshot of the Report File. *Once MAVERIC is finished running, the CPU time data is consolidated and printed to a report file, where it can be easily read and analyzed. The report is a consolidation of the data collected for test case ISS Mean 6DOF of the Ares-I Rev8 series of MAVERIC. CPU usage of each sub-function of "derivatives ( )" is expressed as a percentage of the total CPU time used by "derivatives ()".*

IV. Discussion of Results

After the CPU timer was created, it was put to use to test MAVERIC and search for potential areas for improvement.

A. Timing MAVERIC

The CPU timer was inserted in one of the main functions in MAVERIC called "derivatives ( )". "Derivatives ( )" is a function in MAVERIC that is used to calculate the simulated vehicle's new state, and is called dozens of thousands of times per execution of MAVERIC. The "relative" timer was placed around the whole "derivatives ( )" function, and individual timers were placed around every sub-function within "derivatives ( )". Several runs of MAVERIC were executed with the timers placed in MAVERIC's code, and descriptive statistics were calculated. The results, shown in Figure 6, showed that the top 8 sub-functions made up 83.1% of the total CPU time spent. The top four most costly functions being "Update_Flex ( )", "Update_Mass_Properties ( )", "Update_Aerodynamics ( )", and "Event_Controller ( )".

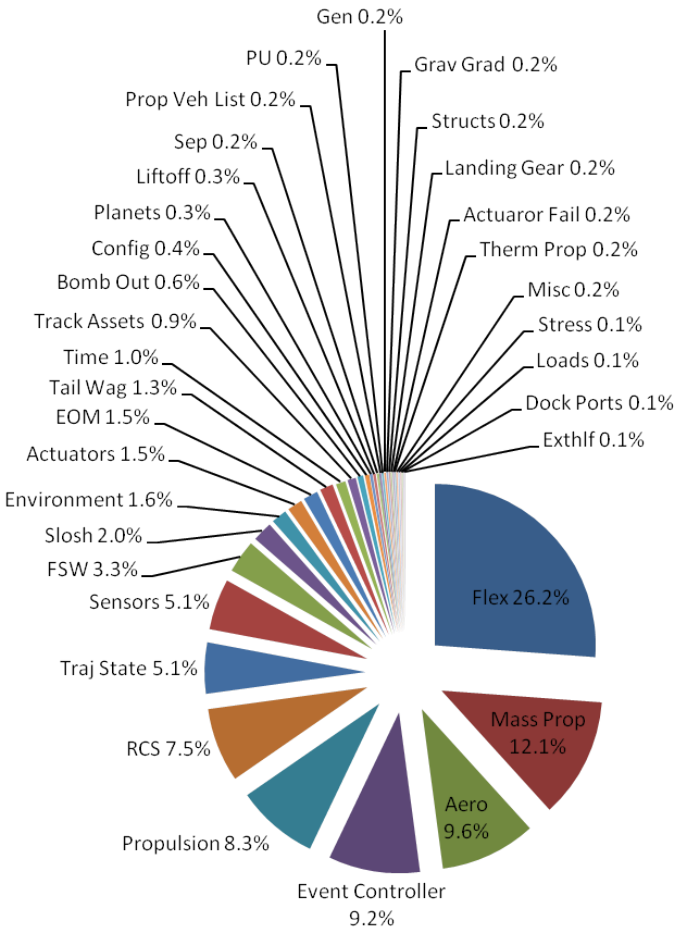

Figure 6. Breakdown of "Derivatives ( )". *After inserting probes in one of the main functions of MAVERIC, data was collected for test case ISS Mean 6DOF of the Ares-I Rev8 series of MAVERIC. CPU usage of each sub-function of "derivatives ( )" is expressed as a percentage of the total CPU time used by "derivatives ()".*

B. Changing the Event Controller

With the help of my mentor, Curtis Zimmerman, an adaption to the function called "Event_Controller ( )" was made to increase efficiency. MAVERIC was run five times before and after the enhancement for statistical data, shown in Table 2 and Table 3, found in the appendix. The averages, listed in Table 1, show that the CPU time used by "Event_Controller ( )" decreased by 36.9%. The overall CPU time used by "derivatives ( )" decreased by 3.2%.

The decrease in CPU time from the change in the Event Controller corresponds to a 1.7 second decrease in real time. For a 2000-count Monte Carlo run, which is performed thousands of times per year during the design phase of vehicle development, the decrease in CPU time corresponds to a 59 minute decrease in real time. This

Averages, Standard Deviations		Event Controller		Percent Decrease
		Old	Enhanced	
"Derivatives ()" Total Cycles	μ	5.36E+10	5.23E+10	2.4%
	σ	1.19E+09	1.43E+09	
	σ / μ	2.2%	2.7%	
Event Controller Total Cycles	μ	4.80E+09	3.03E+09	36.9%
	σ	3.30E+08	2.13E+08	
	σ / μ	6.9%	7.0%	
Percent of "Derivatives ()"	μ	9.0%	5.8%	35.3%
	σ	0.6%	0.3%	
	σ / μ	6.3%	4.8%	

Table 1. Averages, Standard Deviations, and Percent Decrease of CPU Time for the Event Controller. *After enhancing the Event Controller, the percent decrease in CPU Time was calculated for test case ISS Mean 6DOF of the Ares-I Rev8 series of MAVERIC.*

small change in efficiency has led to hours of saved CPU time, which can be used for other tasks, instead of waiting for simulations to finish.

V. Conclusion

As large computer programs grow, it is important to continually enhance them to avoid inefficiency. Having proper tools is critical for any task. This particular tool is especially important to have, because the small change in CPU time attained from enhancement of a function is nearly impossible to accurately measure and assess without a timing function of this type.

Though the program is currently adapted to work with MAVERIC's output system, it can be easily adapted to be used with other programs. With a few small changes, the function can be used to profile most programs in C and C++, as well as most programs written in a compatible programming language.

The CPU timer is very flexible, and is easily implemented. By using a CPU timer, accurate measurements can be made to determine which functions need improvement. By making small enhancements in the program's code to decrease CPU time, hours of time spent waiting for simulations to finish can be saved.

Appendix

Un-Enhanced Event Controller			
Run	CPU Time		%
	Event Controller	"Derivatives ()"	Percent of "Derivatives ()"
1	4.478E+09	5.198E+10	8.6%
2	5.195E+09	5.443E+10	9.5%
3	5.072E+09	5.386E+10	9.4%
4	4.753E+09	5.279E+10	9.0%
5	4.486E+09	5.486E+10	8.2%

Table 2. Run Times for the Un-Enhanced Event Controller. *Data was collected for test case ISS Mean 6DOF of the Ares-I Rev8 series of MAVERIC.*

Enhanced Event Controller			
Run	CPU Time		%
	Event Controller	"Derivatives ()"	Percent of "Derivatives ()"
1	3.301E+09	5.335E+10	6.2%
2	3.193E+09	5.391E+10	5.9%
3	2.797E+09	5.035E+10	5.6%
4	2.975E+09	5.147E+10	5.8%
5	2.876E+09	5.229E+10	5.5%

Table 3. Run Times for the Enhanced Event Controller. *Data was collected for test case ISS Mean 6DOF of the Ares-I Rev8 series of MAVERIC.*

Acknowledgments

I'd like to thank my mentor, Curtis Zimmerman, for all of his help. I would also like to thank NASA Funded National Space Grant College and Fellowship Program for funding this project.

References

Eckel, B. (1993). *C++ Inside & Out.* Berkeley: McGraw-Hill, Inc.

Lippman, S. B. (1998). *C++ Primer.* Upper Saddle River: AT&T.

Rutenberg, G. (2007, September 26). *Profiling Code Using clock_gettime*. Retrieved from Guy Rutenberg Web site.: http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/

Swanson, C. (2010, Spring). *CSci 1113: C++ Programming*. Retrieved from University of Minnesota: College of Science and Engineering: http://www-users.cselabs.umn.edu/classes/Spring-2010/csci1113/index.php?page=day_class_notes